Université Paul Verlaine – Metz

Validation des preuves par récurrence implicite avec des outils basés sur le calcul des constructions inductives

MÉMOIRE

soutenu le 29 Juin 2005

pour le stage de

DEA de l'Université de Metz

(Spécialité Informatique)

par

Cezary Kaliszyk

Composition du jury

	Dominique Méry	Professeur
	Didier Galmiche	Professeur
	Noëlle Carbonell	Professeur
Encadrant :	Sorin Stratulat	Maître de Conférences

Laboratoire d'Informatique Théorique et Appliquée - EA 3097

Mis en page avec la classe thloria.

Abstract

Proving theorems with automated techniques like those based on saturation and implicit induction is becoming a standard practice, since it offers to the users a framework for solving sometimes complicated problems without much human interaction. On the other hand, in the generated proofs the used induction base and step cases are hard to discover and therefore the proofs are hard to read by humans. The proposed solution will be to translate a specification and a proof created by an implicit induction prover into a specification and a proof in the Calculus of Inductive Constructions using explicit induction. The resulted proof script is checked afterwards by the Coq prover. A general translation mechanism and an implementation of it as a module for the Spike prover will be described.

Keywords: Implicit induction, Spike, Calculus of Inductive Constructions, Coq

Résumé

La preuve de théorèmes avec des techniques automatisées, comme celles basées sur la saturation et la récurrence implicite, est devenue aujourd'hui une pratique courrante, car elle offre aux utilisateurs un cadre pour la résolution des problèmes parfois compliqués sans trop d'interaction humaine. En même temps, dans les preuves générées il est difficile de découvrir le(s) cas de base et le(s) pas de récurrence, ce qui rend difficile leur correction par des humains. La solution proposée sera de traduire les spécifications et les preuves générées par un preuveur qui utilise la récurrence implicite dans des spécifications et des preuves dans le calcul des constructions inductives basé sur la récurrence explicite. Le script de la preuve ainsi générée est ensuite vérifiée par l'outil Coq. Un mécanisme général de traduction, ainsi que sa mise en œuvre en tant que module pour le prouveur Spike, seront décrits.

Mots-clés: Récurrence implicite, Spike, Calcul des Constructions Inductives, Coq

Sommaire

Chapter 1

Introduction

Nowadays, formal methods are often used in various areas of research that require more and more sophisticated verification operations. Formal verification is becoming widely used, since the proofs are becoming larger and their soundness is hard to be seen by humans. Formal methods are the only means for checking the correctness of large software programs or important theorems.

Many of these techniques are capable of handling industrial-size examples. Some cases have even been used on a regular basis in industry. The success of formal specifications can be assigned to the ability of system designers to use notations for new methodologies and apply them effectively.

Formal methods are mathematically based languages, techniques and tools for specifying and verifying theorems or properties about systems. Many frameworks have been built upon formal methods to provide to the users the ability to specify, develop, and verify hardware or software systems.

Automated theorem provers for first-order logic are procedures or programs that can be used to show that a given formula (goal) is implied by a first-order theory, usually defined by a finite set of axioms. Automatic theorem proving in first-order logic has been a subject of research for over 30 years, which made it one of the most developed computer science domains.

1.1 Applications of the Automated Theorem Proving

Automated theorem proving is used in many areas of mathematics and computer science, of which the most worth of mentioning are:

- Assisting mathematicians – A first-order prover can be used on its own to relieve a human mathematician from the burden of technically difficult proofs. An example of a proof done with the help of an automated theorem prover is that of the Robbins conjecture [20].

- Formal software verification – A prover may be used to increase the reliability of a safety-critical application and reduce the development cost of complex software systems. Many systems for annotating source code with formal parts were created, to allow the verification by a prover of the code against the model specification. Some examples of projects involving automated theorem proving are [1, 16].

- Automatic software synthesis – Software components may be used to quickly create reliable programs. The components are often augmented with specifications of their functionality forming logical theories. To create a program, one tries to prove the existence of such a program in theory, using the problem domain and the available components. If a proof is found, a program may be extracted from a combination of the used components. An example of such program creation is the Amphion NASA project [19].

- Hardware verification – Most visible application of formal methods. Due to the increasingly complexity of electronic circuits, formal methods like model checking and satisfiability checking are nowadays routinely used for analysing and verifying hardware designs. First-order logic based specifications offer a more powerful language than the propositional logic for specifying hardware models and their behaviour. Usage of automated theorem proving for commercial hardware verification is described for instance in [8].

1.2 Induction-based Proof Techniques

Induction based reasoning is widely used in proving theorems involving unbounded structures, like integers and lists. There are two most known approaches for it:

- Explicit induction – the induction scheme is explicitly stated in the proof. It consists of the application of an inference rule locally in a proof for generating the base and step induction cases. Therefore, the explicit induction can be easily controlled.
- Implicit induction – the validity of the formulas treated as induction hypotheses in the proof can only be determined by analysing the whole proof. The induction principle is globally implemented by all the rules of the inference system.

Both the induction approaches rely on appropriate induction principles, whose soundness is ensured by the existence of a Noetherian order. Defined on a non-empty set of elements, it forbids any infinite strictly decreasing sequence of elements.

1.2.1 Explicit Induction

The most known induction principle is the Peano induction principle to reason on naturals. It has been described by Pascal in 1654, but has been used earlier by the ancient Greeks. It can be soundly extended, under the name of Noetherian principle, for any term structure on which a Noetherian order exists, since for every element the number of it's predecessors up to the minimal element is finite, therefore the unrolled proof is finite.

To prove a property $P(x)$ on any ordered term structure, it is sufficient to show two properties:

- The base case – Show that the property is valid for the smallest element of the structure (in case of naturals it is sufficient to prove $P(0)$).
- The induction step – Show that the property is valid for any given element, if it is valid for its predecessor (in the case of naturals: for any natural k show that $P(k) \Rightarrow P(k+1)$).

The induction is referred to as *explicit*, since in the proof the hypotheses and the conclusions are easily distinguishable. Many existing theorem provers use explicit induction, e.g. NQTHM [7], PVS [23], Coq [9].

1.2.2 Implicit Induction

The *implicit induction* proof technique is an application of the 'Descente Infinie' principle, described later in Chapter 2.

The use of induction in a proof done by implicit induction is global, and the correctness of the formulas used as induction hypotheses can be verified only after the proof is finished. The implicit induction also relies on a Noetherian order, but as opposed to explicit induction, here the order is defined on formulas and not on terms. A given formula can be transformed by an inference rule using induction hypotheses consisting of smaller (or equal) formulas or instances of formulas from the whole proof.

Some examples of provers that use the implicit induction technique are SPIKE [6, 29], QUODLIBET [17] and RRL [18].

1.3 Motivation

Due to the big number of formulas encountered in a proof, finding which formula from an implicit induction derivation can be used as an induction hypothesis is often difficult. This is a reason why such proofs are not done by hand. Since they are automatic, the generated proofs look sophisticated, and verifying their correctness is hard for the user for the following reasons:

- the generated proofs are often very long,
- each application of an inference rule has to be verified to respect the 'Descente Infinie' principle. For some proof steps, the order between different formulas has to be computed, which can be a rather difficult task for the user.

The idea is to check such proofs with more expressive and powerful explicit induction based tools, like Coq [9], known to be able to certify their proofs. On the other hand, a weak point of such tools is their lack of automatization for sophisticated proofs[1].

From a practical point of view, our ultimate goal is to generate sophisticated proofs by Spike that are certified later by Coq.

1.4 The Proposed Solution

Converting a proof generated by an implicit-induction prover to an explicit induction proof will consist of the following parts:

- Theoretical description of two existing implicit and explicit induction proof systems, followed by the description of the translation of the specifications and proofs from the implicit induction system to the explicit induction system..
- Implementation description of a proof translator from Spike to Coq.

1.5 Overview

The rest of the document contains four chapters and one appendix.

Chapter 2 explains the general principle of 'Descente Infinie' and one of its applications, the implicit induction. The prover Spike is presented, together with its inference system.

Chapter 3 describes the Calculus of Inductive Constructions, a theory upon which the prover Coq is based, the prover itself and a part of its language used further in the translator.

Chapter 4 contains the proposed solution, the theoretical possibilities of translating proofs from implicit induction to explicit induction and a module for Spike that translates the specification and the generated proof in a form accepted by Coq.

In Chapter 5, a conclusion and proposed extensions to the framework are described.

Some examples of translated proofs are presented in the Appendix A.

[1]There exist proof techniques (like `auto` in Coq), which can solve very simple queries, but their application to bigger proofs is very limited.

Chapter 2

The Principle of 'Descente Infinie'

The *implicit induction* proofs[2] are applications of the "Descente Infinie" induction principle invented by Fermat in 1659 [32]. The principle states that all the formulas from a possibly infinite set of formulas P are true, if and only if, for any counterexample from P, it exists a smaller counterexample. In the following, by counterexample we understand any ground formula (formula with no variables) that is not true in the logical model.

A "Descente Infinie" proof consists of successive applications of inference rules to a set of conjectures to prove. It can be considered as a set of transitions between states representing sets of conjectures, here denoted by E_0, E_1, ...:

$$E_0 \vdash E_1 \vdash E_2 \vdash \ldots \tag{2.1}$$

Every step of the proof is the result of the application of an inference rule. A 'Descente Infinie' inference system is a collection of such inference rules. It is sound if the minimal counterexamples of all the conjectures are not eliminated during the proof. Therefore, all the conjectures from a proof are true if the proof finishes and no minimal counterexample is detected in its last state.

2.1 The Implicit Induction Proof Technique

To satisfy the condition that no minimal counterexample is in the last state, the implicit induction proofs finish with an empty set of conjectures. During the proof, a conjecture is soundly eliminated by an inference rule if none of its ground instances is a minimal counterexample. One particular case to consider is when it exists a smaller and logically equivalent formula that do not contain minimal counterexamples. Such formulas, which can be any conjecture from the proof, play the role of induction hypotheses. To automatize the process of searching for induction hypotheses, a solution is to keep two sets of formulas in every proof state:

- E – the set of conjectures, as previously.
- H – the set of premises. The elements of H are ancient conjectures, that do not contain minimal counterexamples.

In this case, the search for induction hypotheses is limited only to the formulas from the current set of premises and conjectures.

A proof in the new setting has the form:

$$(E_0, \emptyset) \vdash (E_1, H_1) \vdash \ldots \vdash (\emptyset, H_n) \tag{2.2}$$

i.e. it starts with an empty set of premises and finishes with an empty set of conjectures.

If the inference system that generated such proof is sound, then all the formulas (i.e. conjectures and premises) used in the whole proof are valid.

[2]Also other types of proofs, for example saturation based proofs.

2.2 Application: the SPIKE Theorem Prover

One of the provers that implement the technique of *Implicit Induction* is SPIKE. It was first implemented and described by Adel Bouhoula and Michaël Rusinowitch in [6]. Later, its inference system has been redesigned by Sorin Stratulat. All the inference steps have been generalized as the instances of the abstract inference system A described in [29].

SPIKE specifications include axioms, which are first-order conditional equations and the conjectures are equational clauses[3]. SPIKE is able to check initial consequence relations in many such theories. The Noetherian order on clauses is built starting from the order over the constructor and defined function symbols given in the specification.

SPIKE has been used for the verification of some nontrivial problems, like the validation of an ABR conformity algorithm [26] and of a JavaCard platform [4].

2.2.1 The Inference System

An integral part of a proof system is the inference system. A comparison of available implicit induction inference systems has been done by Sorin Stratulat in his Ph.D. thesis [28].

To perform this comparison, he proposes an abstract inference system A, which provides the widest available induction hypotheses. The A system includes just two rules which take a conjecture, called current conjecture, and replaces it with a new (possibly empty) set of conjectures using induction hypotheses from a content specific to each rule:

- `AddPremise` – adds the current conjecture to the set of premises if it does not contain minimal counterexamples.
- `Simplify` – does not add the current conjecture to the set of premises but allows bigger contents.

Each SPIKE inference rule to be described below is an instance of one of these two abstract rules.

Case Simplify

`case simplify` is a rule permitting simplification of the processed conjecture, due to application of case analysis rule (defined below):

$$(E \cup \{C\}, H) \vdash (E \cup E', H) \tag{2.3}$$

if $E' = CaseAnalysis(C)$.

Definition 1 *If C is an equational clause, p a defined symbol position in C and $\cup_{i=1}^{n}\{P_i \Rightarrow l_1 \rightarrow r_i\}$ a set of conditional rewrite rules, such that for every $i \in [1..n]$, there exists a substitution σ_i, so that after substituting variables in l_i it is identical to s. Then $CaseAnalysis(C[s]_p)$ returns the set $\cup_{i=1}^{n}\{P_i\sigma_i \Rightarrow C[r_i]_p\}$ if in the initial model the logical sum of all P_i with appropriate substitutions, $\bigvee_i P_i\sigma_i$, is valid.*

In the above definition we understand the following by:

substitution – a mapping from variables to ground terms (terms without variables).

positions in a term – natural number sequences, that allow to locate in the tree structure of a term or clause each of its function symbols or variables. $C[s]_p$ means that the clause C contains the term s at the position p.

conditional rewrite rule – an equation over terms characterizing the computation system and having associated a condition under the form of a conjunction of literals. The rewrite rule will be used to replace in a term or clause subterms equal to its left-hand side by its right-hand side if the condition is satisfied.

[3]By an equational clause we understand a disjunction of literals $\neg n_1 \vee \ldots \vee \neg n_k \vee \ldots \vee p_1 \vee \ldots \vee p_j$, where the literals are equations or disequations. Sometimes, it is conveniently represented as an implication between two sets of equations $n_1 \wedge \ldots \wedge n_k \Rightarrow p_1 \vee \ldots \vee p_j$.

Generate

`generate` is the only instance of `AddPremise`. It consists of a case analysis on the domain values of variables subset of the current conjecture C. It takes C and replaces these variables with constructor terms that define their domain under the form of cover substitutions. All the newly obtained conjectures are afterwards processed to make sure that C does not contain minimal counterexamples.

Its formal definition is:

$$(E \cup \{C\}, H) \vdash (E \cup \{\cup_\sigma E_\sigma\}, H \cup \{E\}) \tag{2.4}$$

if for all cover substitution σ of C we have one of the following:

- $C\sigma$ is a tautology and $E_\sigma = \emptyset$,
- $C\sigma$ can be rewritten to C' using axioms, rewrite rules from E that are smaller than C, or smaller or equal rewrite rules from H. Then $E_\sigma = \{C'\}$, or
- $E_\sigma = CaseAnalysis(C\sigma)$

Due to the addition of the current conjecture to the premises, `generate` may be sometimes simulated by an explicit induction rule.

Simplify

`simplify` is the rule that uses inductive rewriting:

$$(E \cup \{C\}, H) \vdash (E \cup \{C'\}, H) \tag{2.5}$$

if C can be rewritten to C' using axioms or smaller or equal rewrite rules from $H \cup E$.

Subsume

`subsume` is a rule eliminating redundant conjectures.

$$(E \cup \{C\}, H) \vdash (E, H) \tag{2.6}$$

if C is subsumed by any axiom or formula from $E \cup H$.

Definition 2 *If C_1 and C_2 are two clauses, we define that C_1 subsumes C_2 if there exists a substitution σ, such that $C_1\sigma$ is a sub-clause of C_2.*

Delete Tautology

`delete tautology` is a rule eliminating trivial conjectures.

$$(E \cup \{C\}, H) \vdash (E, H) \tag{2.7}$$

if C is a tautology, i.e. an equational clause which contains on the right-hand side of the implication an equality of the form $t = t$.

Positive Decomposition

`positive decomposition` rule applies on a clause having in the the right-hand side of the implication an equality with the same free constructor symbol heading the both sides.

$$(E \cup \{f(\vec{s}) = f(\vec{t}) \vee r\}, H) \vdash (E \cup (\cup_{i=1}^{n}\{s_i = t_i \vee r\}), H) \tag{2.8}$$

if f is a free constructor symbol.

Negative Decomposition

`negative decomposition` rule applies on a clause having in the the left-hand side of the implication an equality with the same free constructor symbol heading the both sides.

$$(E \cup \{\neg(f(\vec{s}) = f(\vec{t})) \vee r\}, H) \vdash (E \cup (\vee_{i=1}^{n}(\neg(s_i = t_i) \vee r)\}), H) \tag{2.9}$$

if f is a free constructor symbol.

Positive Clash

The `positive clash` rule eliminates from a clause the equations from the right-hand side of the implication that have different free constructor symbols heading the both sides.

$$(E \cup \{f(\vec{s}) = g(\vec{t}) \vee r\}, H) \vdash (E \cup \{r\}, H) \tag{2.10}$$

if f and g are two different free constructor symbols.

Negative Clash

`negative clash` rule removes the clauses having in the left-hand side of the implication inequations with the both sides headed by distinct free constructor symbols.

$$(E \cup \{\neg(f(\vec{s}) = g(\vec{t})) \vee r\}, H) \vdash (E, H) \tag{2.11}$$

if f and g are two distinct free constructor symbols.

Eliminate Trivial Equation

`eliminate trivial equation` removes equations having the same term on both sides.

$$(E \cup \{\neg(s = s) \vee r\}, H) \vdash (E \cup \{r\}, H) \tag{2.12}$$

Delete

$$(E \cup \{\bigvee_{i=1}^{n} \neg(x_i = t_i) \vee r\}, H) \vdash (E, H) \tag{2.13}$$

If for all i we have: $x_i \notin Var(t_i)$ and $r\rho$ is a tautology, where $\rho = \{x_i \leftarrow t_i | i \in [1..n]\}$.

Autosimplification

The `autosimplification` rule takes a conjecture having on the left-hand side of the implication an equation between a variable and a term, and replaces in the rest of the conjecture all the occurrences of the variable with that term. Example:

$$(E \cup \{x = b \Rightarrow f[x, b] = g[x, b], H\} \vdash (E \cup \{f[b, b] = g[b, b]\}, H) \tag{2.14}$$

Chapter 3

The Calculus of Inductive Constructions

Calculus of Inductive Constructions (CIC) is a type theory derived from the typed λ-calculus.

λ-calculus is a formal language having an expressive power equivalent to Turing machines and recursive functions. All λ-calculus terms express some calculations, but the language is inconsistent; moreover, certain calculations may not finish for some arguments. To fix this, Curry and Church proposed simple-typed λ-calculus, where if a term is defined to have some type it is known to be correct for this type. Later Curry and Howard proposed a mechanism verifying the terms and their equivalent types.

The syntax of simply-typed λ-calculus is very simple. Any type U can be either a simple type S, or a function taking a term of sub-type and returning a term of sub-type. The terms t are either variables x, application of term to a term, or the explicit specification of the type of a term.

A term is only a pre-term until it is correctly typed. The rules of typing terms have been expressed as deduction rules. For a set of terms with their types Γ the deduction rules include typing judgements like $\Gamma \vdash (t : T)$. The Curry-Howard isomorphism is defined as a set of deduction rules on such judgements.

The next step towards its implementation in Coq was the proposition by Coquand and Huet of the Calculus of Constructions [12, 27]. The two newly added concepts are the polymorphic types and the dependent types.

The syntax of Calculus of Constructions is as follows:

$$\begin{array}{rcl} t & ::= & x \;\mid\; (tt) \;\mid\; [x:U]t \;\mid\; (tU) \\ U & ::= & S \;\mid\; X \;\mid\; [X:U]U \;\mid\; (UU) \;\mid\; (X:U)U \;\mid\; (Ut) \\ S & ::= & Prop \;\mid\; Set \;\mid\; Type \;\mid\; TypeSet \end{array}$$

In CIC [10, 11] a syntax for expressing inductive types has been added. This permits operating on them and removes the necessity of `Type` and `TypeSet` in simple types. In CIC every function has to be a total function. This means that for every recursive definition there has to be associated a measure function, providing a means of ensuring that the calculation will finish for every input.

The rules for Curry-Howard isomorphism have been extended to deal with the introduced inductive types, by giving rules for reasoning on inductive types including terms. Therefore, in CIC with every function or theorem definition there is a type associated using the Curry-Howard isomorphism, which ensures the completeness and soundness.

3.1 The Coq System

Coq [9] is a proof assistant based on CIC. It has been basically designed as a syntax for writing CIC terms and verifying that they match defined types (or calculating those types). It has grown into a big set of tools allowing interactive theorem proving.

With every Coq definition and proof step there exists an associated CIC type term. The core system verifies that the element matches the type, which guarantees the totality of all the defined functions and also the soundness of all the proofs.

A proof written in the Coq system is mechanically checked by the kernel of the system, to verify if its type matches its definition. Coq allows a user to define sets and their properties by the means of: functions (both general recursive functions and functions defined by the axioms concerning them) and theorems.

Coq is an interactive system and allows the user to interactively develop proofs. The system allows the user to define only objects that are correct in the *Calculus of Inductive Constructions*, in particular:

- All the functions have to be total, i.e. they have to be defined for all the elements of the set upon which they are defined.
- For every recursive function, a measure function has to be provided to assure the system that the calculation of the function will finish for all the values of its arguments.
- For every theorem, there should exist an associated proof, which will be verified to be correct w.r.t. the Curry-Howard isomorphism in CIC.

The Coq system also includes a mechanism for the automatic generation of certified programs from the proofs of their specifications.

But for `coqtop`, the standard Coq interactive toplevel, various user interfaces have been created to allow better interaction with the user, e.g. CoqIde (included in the Coq distribution) and ProofGeneral [2].

The Coq system has been used for the verification of many non-trivial problems, like the JavaCard bytecode specification [3] or the four colours theorem [14].

3.2 Coq Tactics and Tacticals

The Coq inference system is defined by a set of tactics and tacticals (i.e. strategies to apply tactics). Every tactic is a rule describing how to convert a goal to a new set of goals, that will be easier (hopefully) to type. In every proof step, a tactic is applied to one of the goals.

Since Coq is a big system, it includes many tactics and tacticals, and the full list is described in [9]. The following subsections describe the ones that will be used in this document or the translator.

3.2.1 The `Intro` Tactic

This tactic applies to a goal which is a product[4].

If the current goal is a dependent product `forall x:T, U` then `intro` puts `x:T` in the local context and replaces the goal with its subgoal `U`.

If the goal is a non dependent product of the type `T -> U`, then the tactic puts in the local context `Hn:T`[5]. The optional index n is such that Hn is a fresh identifier, and the new subgoal is `U`.

3.2.2 The `Induction` Tactic

The `induction` tactic applies to any goal. The type of the argument term must be a constant of an inductive type. The tactic generates subgoals, one for each possible form of term, i.e. one for each constructor of the inductive type.

The `induction` tactic automatically replaces all the occurrences of `term` in the conclusion and the hypotheses of the goal. It automatically adds induction hypotheses (named `IHn1`) to the context.

There is one particular case, that will have to be treated differently in the conversion system. If `term` is an identifier denoting a quantified variable of the conclusion of the goal, then `induction ident` behaves like: `intros ident; induction ident`.

[4]The `Intro` tactic also applies to goals which start with a `let` binder, but they will not be used in this work.

[5]It works different, if `T` is a `Type`, it will not be used in this document.

3.2.3 The `Destruct` Tactic

The `destruct` tactic is used to perform case analysis without recursion. It applies to any goal and the type of `term` must be an inductive type. Its behaviour is similar to `induction`, but no induction hypothesis is generated.

If `term` is an identifier denoting a quantified variable of the conclusion of the goal, `destruct` behaves like induction, which is: `intros ident; destruct ident`.

3.2.4 The `Double Induction` Tactic

The `double induction` id_1 id_2 tactic applies to any goal. If variables id_1 and id_2 of the goal have an inductive type, then this tactic performs induction on both of these variables. This rule stores two induction hypotheses, one as the initial goal, and one for the intermediate cases.

A double induction rule will be used, if in a Spike generated proof, there is a `generate` rule applied to two variables.

For instance, if the current goal is `forall n m:nat, P n m` then, `double induction n m` will generate all the four cases with their respective inductive hypotheses.

3.2.5 The `Rewrite` Tactic

The `rewrite` tactic applies to any goal. The type of term must have the form:

$$(x_1 : A_1) \dots (x_n : A_n) term_1 = term_2. \tag{3.1}$$

Then `rewrite term` replaces all the occurrences of $term_1$ by $term_2$ in the goal. If during the process of unifying the terms $term_1$ and $term_2$ there are variables x_i that are instantiated, then certain types $A_1, \dots, A_n$ become new subgoals.

There exist many variants of `rewrite`, the only nonstandard used by the translator is: `rewrite <- term`, which employs the equality $term_1 = term_2$ from right to left. In certain cases it is used to rewrite a goal by applying an induction hypothesis saved by Coq in the 'left to right' format.

The `rewrite` tactic is one of the mostly used in the proofs generated by the translator, that employs axioms, already proved lemmas and induction hypotheses.

3.2.6 The `Apply` Tactic

The `apply` tactic applies to any goal. It tries to match the current goal against the conclusion of the type of `term`. If it succeeds, then the tactic returns as many subgoals as the number of non-dependent premises of the type of the term. The `apply` tactic relies on first-order pattern-matching with dependent types.

If a second-order pattern-matching problem is involved, in order to use the `apply` tactic, the problem has to be transformed into a first-order one using the `pattern` tactic.

3.2.7 The `Trivial` Tactic

The `trivial` tactic applies to any goal. It tries to check if the goal is in the current assumptions and if not, it tries to match the list of tactics with the head of the goal, and apply these tactics.

It is typically used to solve trivial equalities, like $X = X$[6].

3.2.8 The `Tauto` Tactic

This tactic implements a decision procedure for propositional calculus based on the calculi LJT* of Roy Dyckhoff [15]. `tauto` unfolds negations and logical equivalences but does not unfold any other definitions.

[6] `trivial` is a special variant of Prolog-like resolution provided by `auto` tactic, but applied with maximal depth 0.

3.2.9 The `Symmetry` Tactic

The `symmetry` tactic applies to any equality.

It replaces a goal of the form:

$$P = Q \tag{3.2}$$

with:

$$Q = P \tag{3.3}$$

3.2.10 The `Discrimination` Tactic

The `discriminate` tactic proves any goal from an absurd hypothesis stating that two structurally different terms of an inductive set are equal. For example, from the hypothesis (S (S O))=(S O) we can derive by absurdity any proposition. Assume that the argument is a hypothesis of type $term_1 = term_2$ in the local context, $term_1$ and $term_2$ being elements of an inductive set. To build the proof, the tactic traverses the normal forms of $term_1$ and $term_2$ looking for a couple of subterms u and w, placed at the same positions and whose head symbols are two different constructors. If such a couple of subterms is found, then the proof of the current goal succeeds, otherwise the tactic fails.

3.2.11 The `Omega` Tactic

There are many additional tactics, that have been written for Coq, one of which is `omega`, written by Pierre Crégut [24].

omega solves a goal in Presburger arithmetic, i.e. universally quantified formulas made of equations and inequations. Equations may be specified either on the built-in types `nat` of natural numbers or on the type `Z` of binary-encoded integer numbers. Formulas on `nat` are automatically injected into Z. The procedure may use any hypothesis of the current proof session to solve the goal.

`omega` handles multiplication, but only the goals having at least one of the two factors of the products as a constant are solvable. This restriction is meant in Presburger arithmetic.

Since most of Spike reasoning modules find inconsistencies on naturals, therefore the `omega` tactic will be used to solve in Coq the goals that were proved to be inconsistent by these reasoning modules.

3.2.12 The `Try` Tactical

The `try` tactical applies the tactic given as the argument and if it doesn't succeed it doesn't produce any error. Since the Spike version of the translated tactic does not implement this, it may be sometimes applied a number of times consecutively. To omit this error, all the translations of such rules will use the `try` tactical before the tactic.

The `try` tactical will be most often used, when translating a rewrite or normalization of Spike terms using a number of `rewrite`s applied to the same goal. Storing the last rewrite is not sufficient, since the order of using `rewrite` rules is sometimes not subsequent. It may be so, in case where rule A is used earlier by Spike than rule B, but B may allow rewriting using rule A. In such case the generated rewrites may appear in order: $E_i \vdash_A E_{i+1} \vdash_B E_{i+2} \vdash_A E_{i+3}$ and the last translated rewrite may be already performed by the first one.

Chapter 4

The Proposed Solution

4.1 Translating the Inference System

Any given instance of the abstract inference system includes rules as instances of the abstract inference rules. As the translation problem in general is difficult (see the next subsection for an example), we have restricted to consider only the following particular but rather common case.

The translation of rules being instances of `Simplify` corresponding to deductive reasoning are relatively simple. Therefore, since it does not use any inductive hypothesis, its local logical meaning in the proof is the same as in an explicit induction proof. Assuming that the language of the prover to which the proof is translated is powerful enough (as it is the case of SPIKE and COQ), it can be directly rewritten to the language of the other prover. Instances of `Simplify` using induction hypotheses are treated only when the induction hypotheses can be represented by an explicit induction schema.

The rules being instances of `AddPermise` rule are expected to introduce induction hypotheses which will later be used by the rules of both types. The first step of recreating a proof in explicit induction is the finding of hypotheses that have been finally used in the proof. The locations in the proof where those hypotheses were introduced and finally used will be examined.

If a hypothesis is introduced in implicit induction for just one of the conjectures, it can be used afterwards by the subproofs of all of them. First we will show how to translate a proof, where all the hypotheses introduced by instances of `AddPermise` are used in the proof only by the processed conjecture.

If a hypothesis is introduced using a SPIKE-like `generate` rule it can be directly translated into an explicit induction schema on the term which was subject to case analysis. If the rule does not concern case analysis, then the rewrite rule using this rule is unidirectional, so the proof will never return to the same clause, unless a case analysis will be made. In the latter case we can convert this case analysis to an explicit induction step and, therefore, obtain the induction hypothesis in the place where it will be used.

4.1.1 Non-translatable Proofs

Sometimes, the translation is impossible if the hypothesis is used in the proofs of other conjectures. A very simple example of a proof that is not translatable (the rewrites after `generate` steps and a number of steps between the two `generate`s have been omitted to increase its readability):

$$\begin{array}{rcl} (\{C[x,y]\}, & \emptyset) & \vdash \\ (\{C[0,y], C[S(x),y]\}, & \{C[x,y]\} & \vdash \ldots \vdash \\ (\{C[0,y], C[S(x),0], C[S(x),S(y)]\} & \{C[x,y], C[S(x),y]\} & \end{array}$$

The closest translation of this proof to explicit induction might be (here the base steps, rewrites and a number of steps of proof have been omitted):

$$\begin{array}{cl} C[x,y] & \vdash \\ C[x,y] \Rightarrow C[S(x),y] & \vdash \ldots \vdash \\ (C[x,y] \Rightarrow C[S(x),y]) \Rightarrow (C[x,S(y)] \Rightarrow C[S(x),S(y)]) & \end{array}$$

The proof cannot be continued, since when proving $C[S(x),S(y)]$ we cannot use the hypothesis $C[x,y]$, whereas in the implicit version we can.

If there would be no inference steps between the two `generate`s, the proof would be simply convertible to a translatable one. Such proofs are normally a single `generate` on many variables, and are translatable like above (the exact translation is described in Subsection 4.3.3) since we can use one induction step, and therefore the hypothesis is applied to the conjecture for which it has been defined.

4.2 Implementation

The solution has been implemented as a new module in SPIKE. This way the parser built into the prover is used to parse the specification. Also the SPIKE internal representations of clauses are used for translating and outputting them to COQ specifications.

When a proof in SPIKE is completed in the internal representation there is no history recorded. Therefore, the implementation has to record all the operations that succeeded.

Some operations in SPIKE represent a number of simple operations. For example, a `normalize` operation consists of many `rewrite` operations applied to a conjecture and one `generate` consists of an `induction` and `rewrite` operations.

A new option `-coq` has been added to the SPIKE prover that, if activated, stores all the proof steps and after the proof has been completed, outputs the complete COQ specification and proof script to a file.

4.2.1 Defining Types

All the inductive types are defined using the `Variable` instruction. The constructors of all the types containing constructors are collected together and are output together in the COQ specification as one `Inductive`.

The only exception is the boolean type, which in SPIKE is defined as having two constructors `true` and `false`, whereas in the corresponding COQ specification it has to be defined twice:

- Once as a normal inductive type `sp_bool` which is a `Set` containing the two appropriate constructors.
- Once as a `Prop` to be further able to use COQ rules which operate on clause negation, like the `Classical` reasoning module.

4.2.2 Defining Lemmas and Conjectures

All the lemmas listed in a SPIKE specification, as well as the goal of all the conjectures to prove, are rewritten to COQ by taking into consideration the type of the expression upon which the clause is defined:

- If a clause is defined on boolean clauses and includes equations of the type $P(x) = \mathit{true}$ it has to be rewritten to only include the proposition $P(x)$.
- If a clause is defined on boolean clauses and includes equations of the type $P(x) = \mathit{false}$ it has to be rewritten to only include the proposition $\neg P(x)$.
- If a clause is defined on clauses of other types, the equations are preserved.

4.3 The Implemented Rules

4.3.1 Conditional Rewrite

After performing a conditional rewrite operation, the current clause is normalized. The SPIKE normalizing module tries to match available rewrite rules both from the axioms, lemmas, and the content associated to each inference rule. For every successful rewrite operation the translator remembers the used rules which are output in the COQ proof script.

The rewrite rules using axioms and lemmas are rewritten using the COQ rewrite, while the rules, with which the rewrite rules used as induction hypotheses are used, will be described further.

4.3.2 `generate` on One Variable

The `generate` rule, in its most simple variant, instantiates only one variable of the current clause.

In the generated COQ proof this is obtained using `induction` on the variable, upon which the induction is based in case of it being an inductive type.

The COQ induction tactic requires that the variable, upon which we want to call the induction, has to be free of quantifiers. That is why before calling the induction all the variables existing in the proposition are freed using the `intro` tactic.

Since the SPIKE `generate` has to assure that at the end all the obtained clauses are lesser than the original w.r.t. the order on clauses, the `generate` rule includes rewriting of all the clauses and succeeds only if rewriting of all the clauses is possible.

Therefore, the translator should add new rewrite operations, as described in 4.3.1.

4.3.3 `generate` on More Variables

If a `generate` rule is called on two variables, and all the constructors of both those variables are used, the translation module utilizes the `double induction` COQ tactic. To obtain the same induction hypotheses as SPIKE stores in the set of premises, the order of the variables passed to the tactic is opposite from the order defined in the SPIKE specification. The `intro` tactic has to be called both before and after the `double induction` tactic for specified variables. At the end, each the newly obtained clauses has to be rewritten, as in the `generate` rule.

Translating a `generate` on more than two variables to COQ is possible in an analogical way, using the general `Functional Scheme` command, but since it is very rare, it has not been implemented in the translator.

4.3.4 Total Case Rewriting

A `total case rewriting` analyzes a defined function symbol in a conjecture. It looks for the axioms defining this symbol, that are able to perform rewrite operations with conditional rules.

The translation in COQ of this rule has to include the verification of the disjunction of the conditions of these rules, which is not verified in SPIKE[7], but provided in the specification. The basic step of such translation includes a `cut` using a logical sum of all the possible cases.

In COQ this tactic creates as many goals as all the possible cases and a goal, which states that a possible case has to be true.

To delete the last added goal, an application of the COQ `NNPP` theorem from Classical module has to be used, which proves a clause using double negation. Next, to prove that a negation of all the possible cases is impossible, it is sufficient to apply `tauto`.

For all the obtained clauses we need to move the single logical statement from the goal part to assumptions, to further divide it. To do it, first `intros All` is used, just to do `elim All`. Then the goal clause is replaced with all the goal clauses and the possible cases are moved back from hypotheses to goals.

[7]thanks to the strongly sufficient completeness property of SPIKE specifications.

Now, since all the rewrites required some constraints, we move those constraints into assumptions using `intros Constraint` for every of the new clauses. SPIKE now performed a rewrite on each new conjecture, and in the COQ proof we now translate them to appropriate rewrites as previously. After a rewriting is completed on a conjecture, a new goal is produced, which requires the verification of the correctness of the constraint. To delete all the new additional goals, an `apply Constraint` tactic is used.

4.3.5 `generate` with `Total Case Rewriting`

Often the `generate` rule uses a case analysis rule (see the definition of `generate`) where for a given variable of some type, not only all the constructors of this type are considered, but also some other constraints given on the variable are also considered in order to perform a rewrite.

In such a case, a more complicated proof, including the verification that all the given constructions fill out all the possible cases, has to be made in COQ. In order to achieve this, after the usual `intros` and `induction`, the following two different approaches are considered:

- For the cases generated by constructors of inductive types: a simple rewrite or apply is used (like for the simple inductive type based induction)
- For the cases generated by adding constraints to inductive types: every clause has to be processed like in Subsection 4.3.4 to generate all the possible cases. The rewrite rule applied by SPIKE has to be applied as the rewrite rule used in the translation of `total case rewriting`.

4.3.6 Negative Decomposition

A `negative decomposition` is a rule which transforms a conjecture having an disequation of the same free constructor symbol on both sides in the assumptions, and disassembles those free constructor to disequations of its subterms.

A translation of `negative decomposition` is done, by introducing the rule as a COQ hypothesis, by using `intro`, then calling `discriminate` on the hypothesis to disassemble and move it back to the goal part.

4.3.7 Negative Clash

`negative clash` rule eliminates the clauses having distinct free constructor symbols heading the both sides of an equation from the left-hand side of the implication.

A translation of `negative clash` is done by separating a conjecture of above type from the ones to be proved with applying the `discriminate` tactic to the equation.

4.3.8 Auto Simplification

This rule is translated by firstly introducing the equation between a variable and a term in the hypotheses part by using `intro rule`, then by rewriting the clause using this rule that replaces the variable by the term, and finally by removing the equation from the hypotheses set.

4.3.9 Positive Decomposition

The `positive decomposition` rule removes the identical constructors heading the both sides of an equation.

The translator ignores `positive decomposition` rules; therefore, the rewrite rules will operate on more complicated equalities and the `trivial` tactic will remove a more complicated tautology.

4.3.10 Tautology Delete

Every `tautology delete` rule is rewritten as a `trivial` tactic.

4.3.11 Delete Subsumption

The `delete subsumption` rule is treated in two possible different ways:

- The clause which was removed form the proof is subsumed by one of the lemmas. In this case it is replaced by an application of this lemma (`rewrite` followed by `trivial` or `apply`, as described further).
- The clause is subsumed by a clause from the content of the rule. In the latter case it is translated to an application of the hypothesis.

An application of a given rewrite rule depends on the state of this rewrite rule:

- If this rule is an equivalence then it can be rewritten using the Coq `rewrite` tactic. Afterwards we get a clause containing a tautology, so we eliminate it with `trivial`.
- If the rule is a logical rule, then `apply` can be used to remove it from the set of goals.

4.4 Reasoning Modules

The reasoning modules implemented in Spike that have been tested as part of this research are the ones reasoning on naturals.

Theomega tactic, which solves automatically all the goals in Presburger arithmetic, has been used to solve all the goals solved by the arithmetic reasoning module of Spike.

4.5 Implementation Issues

4.5.1 Clause Numbering

In Coq, the goals are numbered with positive integers that are unique. In Spike, the clauses are numbered w.r.t. the order they are generated, therefore, a part of the proof translation tool must be a module renumbering clauses.

Each time a new clause is produced by Spike, the clause renumbering routine is called in the translator.

4.6 Other Inference Systems

The following 'Descente Infinie' inference system has been proposed in [30]. It has the particularity that the premises are not added by a `generate`-like rule (here denoted by `expand`) but by the rewrite rules:

- `Delete identity`
  $(E \cup \{t = t\}, H) \vdash^P (E, H)$
- `Rewrite`
  $(E \cup \{e\}, H) \vdash^P (E \cup \{\phi\}, H \cup \{e\})$
  where ϕ is a conjecture obtained from e by rewriting.
- `Expand`
  $(E \cup \{C\}, H) \vdash^P (E \cup \Phi, H)$
  where Φ=$\{C\sigma \mid \sigma$ is a cover substitution of $C\}$.

Contrary to Spike, the `Expand` is an instance of `Simplify`, and `Rewrite` is an instance of `AddPremise`. Starting from the specification that defines the addition over the naturals:

- $0 + x \rightarrow x$
- $S(x) + y \rightarrow S(x + y)$

a possible proof of $x + 0 = x$ could be:

$$\begin{array}{rcl}
(\{x+0=x\}, & \emptyset) & \vdash^{P}_{Ex} \\
(\{0+0=0, S(x')+0=S(x')\}, & \emptyset) & \vdash^{P}_{R}\ (\mathit{twice}) \\
(\{0=0, S(x'+0)=S(x')\}, & \{0+0=0, S(x')+0=S(x')\}) & \vdash^{P}_{D} \\
(\{S(x'+0)=S(x')\}, & \{0+0=0, S(x')+0=S(x')\}) & \vdash^{P}_{Ex} \\
(\{S(0+0)=S(0), S(S(x'')+0)=S(S(x''))\}, & \{0+0=0, S(x')+0=S(x')\}) & \vdash^{P}_{R} \\
(\{S(0)=S(0), S(S(x'')+0)=S(S(x''))\}, & \{\ldots, S(x')+0=S(x')\}) & \vdash^{P}_{R} \\
(\{S(0)=S(0), S(S(x''))=S(S(x''))\}, & \{\ldots\}) & \vdash^{P}_{D}\ (\mathit{twice}) \\
(\emptyset, & \{\ldots\}) &
\end{array}$$

To translate it into a Coq proof script, it is sufficient to check which induction hypotheses from the premises have been used. In this proof the only clause that has been used is $S(x')+0 = S(x')$. Therefore, the expand of this rule has to be translated as `induction`, whereas the expand of $x + 0 = x$ can be translated to a `destruct`.

Since at the time of writing this report, this inference system was not yet implemented, the proof has been done manually. A translated version of it has been included in the Appendix.

4.7 Related Works

Judicaël Courant describes in [13] a system that allows to verify an implicit induction proof by Coq, by specifying the order on clauses and translating the generated proof using the defined induction. The proof translation presented there does not use explicit induction; instead, it defines the order on clauses in Coq and gives a Coq specification with the exact proof in implicit induction as generated by Spike. The approach differs from the one presented here also by treating only the inference system of Spike (in a version as it was by Bouhoula [6]) and not the abstract inference system A.

In his Ph.D. thesis and the following publications [22, 21], Nguyen Quang Huy describes a framework for integrating the Coq rewriting with a rewriting-based programming environement ELAN. He presents a translation mechanism from an ELAN proof to a verified proof in Coq.

Chapter 5

Conclusions and Future Works

A translation mechanism from implicit to explicit induction proofs has been described. The implicit induction prover SPIKE has integrated a working module for translating its specifications and proofs into COQ specifications and proof scripts, respectively.

We expect that the translator work for many specifications; even if it is limited to specifications where the hypotheses are used only in the proofs of the conjectures for which they have been introduced, we believe that this is not a serious restriction since other types of proofs are rare. For example, the translator is not yet able to handle proofs of theorems concerning mutually recursive functions. If the inductions concerning two such functions can be joined, than it is possible to use only one induction and afterwards separate the proofs back again.

The translator could be extended by implementing other SPIKE inference rules, like `congruence closure`. The translation of some rules can be improved. For example, the translation of `generate` can be extended to work for more than two variables, using more general schemes than `induction` and `double induction`.

As mentioned before, there exist proofs done by SPIKE which may manipulate hundreds of thousands of clauses (like those concerning the JavaCard Platform verification [4]). In the future, we expect to translate such important proofs into COQ scripts. The current implementation of the translator is limited by: firstly, the lack of ability to work with parametrized specifications, and secondly, the fact that the translator has to memorize all the processed clauses and the places where a used induction hypothesis has been generated. Therefore, the used memory space becomes important; after about 30 minutes of processing the Gilbreath Card Trick proof [5] with the current translator, the prover used about 500MB of memory and the garbage collector started to use the most of the computing time. There is still place for improvements: one solution is to implement the translator as a two-pass process; firstly storing the hypotheses and afterwards memorizing the place where they have been created.

The final goal of the theoretical research would be to show that any proof generated by any instance of the `AddPermise` and `Simplify` abstract inference rules can be translated into an explicit induction proof.

Bibliography

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In Manuel Ojeda-Aciego, Inma P. de Guzmán, Gerhard Brewka, and Luís Moniz Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, volume 1919 of *LNCS*, pages 21–36. Springer-Verlag, October 2000. URL: `ftp://ftp.cs.chalmers.se/pub/users/reiner/jelia.ps.gz`.

[2] D. Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis*, pages 38–42. Laboratory for Foundations of Computer Science, Springer, 2000. LNCS 1785.

[3] G. Barthe, G. Dufay, L. Jakubiec, S. Melo de Sousa, and B. Serpette. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 302–319. Springer Verlag, 2001.

[4] G. Barthe and S. Stratulat. Validation of the JavaCard platform with implicit induction techniques. In *14th International Conference on Rewriting Techniques and Applications (RTA'2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 337–351, 2003.

[5] A. Bouhoula. *Preuves automatiques par récurrence dans les théories conditionnelles.* PhD thesis, Université Nancy I, March 1994.

[6] A. Bouhoula and M. Rusinowitch. *SPIKE-User Manual*, December 1995.

[7] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[8] K. Claessen, R. Hahnle, and J. Martensson. Verification of hardware systems with first-order logic, 2002.

[9] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0.* INRIA-Rocquencourt, January 2005.
http://coq.inria.fr/doc-eng.html.

[10] T. Coquand. *Une théorie des constructions.* PhD thesis, Université Paris VII, 1985.

[11] T. Coquand. Metamathematical investigations of a calculus of constructions. *Academic Press*, 31:pp. 91–122, 1989.

[12] T. Coquand and G. P. Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. In *Logic Colloquium*, pages 123–146. Elsevier, 1985.

[13] Judicaël Courant. Proof reconstruction (Preliminary version).

[14] K. Devlin. Devlin's angle: Last doubts removed about the proof of the four color theorem., Jan. 2005.

[15] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807, 1992.

[16] P. Graf. Substitution tree indexing. Technical Report MPI-I-94-251, Saarbruecken, 1994.

[17] J.Avenhaus, U. Kühler, T. Schmidt-Samoa, and C-P. Wirth. How to prove inductive theorems? QUODLIBET! In F. Baader, editor, *CADE*, volume 2741 of *Lecture Notes in Computer Science*, pages 328–333. Springer, 2003.

[18] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[19] M. Lowry, A. Philpot, T. Pressburger, I. Underwood, R. Waldinger, and M. Stickel. Amphion: Automatic programming for the NAIF toolkit. *NASA Science Information Systems Newsletter*, 31:22-25, 1994.

[20] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[21] Q.-H. Nguyen. Certifying term rewriting proofs in ELAN. *Electr. Notes Theor. Comput. Sci.*, 59(4), 2001.

[22] Q.-H. Nguyen and C. Alvarado. ELAN for equational reasoning in Coq. In INRIA, editor, *2nd Workshop on Logical Frameworks and Metalanguage - LFM'00, Santa Barbara, USA*. J. Despeyroux, Jun 2000. ISBN 2-7261-1166-1.

[23] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, 1992.

[24] William Pugh. The Omega test: a fast and practical integer programming alghoritm for dependant analysis. *Comm. of the ACM*, August 1992.

[25] A. Riazanov. *Implementing an Efficient Theorem Prover.* PhD thesis, University of Manchester, 2003.

[26] M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical verification of an ideal incremental ABR conformance algorithm. *Journal of Automated Reasoning*, 30(2):153–177, 2003.

[27] J. P. Seldin. Coquand's calculus of constructions: A mathematical foundation for a proof development system. *Formal Asp. Comput.*, 4(5):425–441, 1992.

[28] S. Stratulat. *Preuves par récurrence avec ensembles couvrants contextuels. Applications à la vérification de logiciels de télécommunications.* PhD thesis, Université Henri Poincaré, Nancy I, 2000.

[29] S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32(4):403–445, 2001.

[30] S. Stratulat. Automatic "Descente Infinie" induction-based reasoning. (submitted), 2005.

[31] Sorin Stratulat. Lecture support for "Formal Specifications" course. Paul-Verlaine University, Metz.

[32] C.-P. Wirth. Descente Infinie + Deduction. *Logic Journal of the IGPL*, 12(1):1–96, 2004.

Appendix A

Some Translation Examples

A.1 add.spike

The SPIKE specification:

```
sorts : nat ;
constructors :
  0  :      -> nat;
  S_ : nat -> nat;
defined functions:
  _+_ : nat nat -> nat;
axioms:
  0 + x = x;
  S(x) + y = S(x+y);
```

is translated to:

```
Inductive sp_nat : Set :=
  | sp_0 : sp_nat
  | sp_S : sp_nat -> sp_nat
.
Variable sp_plus : sp_nat -> sp_nat -> sp_nat.
Axiom sp_axiom_2 : forall u1 : sp_nat, (sp_plus sp_0 u1) = u1.
Axiom sp_axiom_3 : forall u2 u1 : sp_nat, (sp_plus (sp_S u1) u2) = (sp_S (sp_plus u1 u2)).
```

The proofs of all the conjectures included in SPIKE specification are automatically translated[8] to the following COQ specification:

```
Lemma sp_lemma_4 : forall u1 : sp_nat, (sp_plus u1 sp_0) = u1.
  intros.
  induction u1.
  1:try rewrite sp_axiom_2.
  2:try rewrite sp_axiom_3.
  1:trivial.
  1:rewrite IHu1.
  1:trivial.
Qed.
Lemma sp_lemma_5 : forall u2 u1 : sp_nat, (sp_plus u2 (sp_S u1)) = (sp_S (sp_plus u2 u1)).
  intros.
  induction u2.
```

[8]The above is a cut and paste from the generated proof, only line breaks for the too long lines have been added manually.

```
    1:try rewrite sp_axiom_2.
    2:try rewrite sp_axiom_3.
    1:try rewrite sp_axiom_2.
    1:trivial.
    1:rewrite IHu2.
    1:try rewrite sp_axiom_3.
    1:trivial.
  Qed.
  Lemma sp_lemma_6 : forall u2 u1 : sp_nat, (sp_plus u1 u2) = (sp_plus u2 u1).
    double induction u2 u1.
    1:intros.2:intros.3:intros.4:intros.
    1:try rewrite sp_axiom_2.
    2:try rewrite sp_axiom_2.
    3:try rewrite sp_axiom_3.
    4:try rewrite sp_axiom_3.
    1:try rewrite sp_axiom_2.
    1:trivial.
    1:try rewrite sp_axiom_3.
    1:try rewrite sp_lemma_4.
    1:trivial.
    1:try rewrite sp_lemma_4.
    1:try rewrite sp_axiom_2.
    1:trivial.
    1:try rewrite sp_lemma_5.
    1:try rewrite sp_axiom_3.
    1:try rewrite sp_lemma_5.
    rewrite H0.
    trivial.
  Qed.
  Lemma sp_lemma_7 : forall u3 u2 u1 : sp_nat,
      (sp_plus (sp_plus u1 u2) u3) = (sp_plus (sp_plus u3 u2) u1).
    intros.
    induction u1.
    1:try rewrite sp_axiom_2.
    2:try rewrite sp_axiom_3.
    1:try rewrite sp_lemma_4.
    2:try rewrite sp_axiom_3.
    2:try rewrite sp_lemma_5.
    rewrite sp_lemma_6.
    trivial.
    rewrite IHu1.
    trivial.
  Qed.
```

The whole above specification is verified by Coq.

A.2 Translating the Proof from Section 4.6

```
Inductive mnat : Set :=
  | Ms : mnat -> mnat
  | Mo : mnat
.
Variable mplus : mnat -> mnat -> mnat.
Axiom M1 : forall x : mnat, mplus Mo x = x.
Axiom M2 : forall x y : mnat, mplus (Ms x) y = Ms (mplus x y).
Lemma L1 : forall x : mnat, mplus x Mo = x.
  destruct x.
  induction x.
  rewrite M2.
  rewrite IHx.
  trivial.
  rewrite M2.
  rewrite M1.
  trivial.
  rewrite M1.
  trivial.
Qed.
```